

CA3

March 8, 2023

```
[1]: # Import libraries
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import numpy as np
import itertools
import argparse
import sys
import time
from sklearn import preprocessing
import pandas as pd
import os
os.environ['TF_CPP_MIN_LOG_LEVEL']='3'

[2]: ## Preprocessing of data
# Function to load data

def get_power_data():
    """
    Read the Individual household electric power consumption dataset
    """

    # Assume that the dataset is located on folder "data"
    data = pd.read_csv('household_power_consumption.txt',
                       sep=';', low_memory=False)

    # Drop some non-predictive variables
    data = data.drop(columns=['Date', 'Time'], axis=1)

    #print(data.head())

    # Replace missing values
    data = data.replace('?', np.nan)

    # Drop NA
    data = data.dropna(axis=0)

    # Normalize
    standard_scaler = preprocessing.StandardScaler()
```

```

np_scaled = standard_scaler.fit_transform(data)
data = pd.DataFrame(np_scaled)

# Goal variable assumed to be the first
X = data.values[:, 1:].astype('float32')
y = data.values[:, 0].astype('float32')

# Create categorical y for binary classification with balanced classes
y = np.sign(y+0.46)

# Split train and test data here: (X_train, Y_train, X_test, Y_test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)
no_class = 2 #binary classification

return X_train, X_test, y_train, y_test, no_class

```

```

[3]: X_train, X_test, y_train, y_test, no_class = get_power_data()
print("X,y types: {} {}".format(type(X_train), type(y_train)))
print("X size {}".format(X_train.shape))
print("Y size {}".format(y_train.shape))

# Create a binary variable from one of the columns.
# You can use this OR not

idx = y_train >= 0
notidx = y_train < 0
y_train[idx] = 1
y_train[notidx] = -1

```

```

X,y types: <class 'numpy.ndarray'> <class 'numpy.ndarray'>
X size (1536960, 6)
Y size (1536960,)

```

```

[4]: # Sigmoid function
def sigmoid(x, derivative=False):
    sigm = 1. / (1. + np.exp(-x))
    if derivative:
        return sigm * (1. - sigm)
    return sigm

# Define weights initialization
def initialize_w(N, d):
    return 2*np.random.random((N,d)) - 1

# Fill in feed forward propagation
def feed_forward_propagation(X, y, w_1, w_2, w_3, lmbda):
    # Fill in
    layer_0 = X

```

```

layer_1 = sigmoid(np.dot(layer_0, w_1), derivative=False)
layer_2 = sigmoid(np.dot(layer_1, w_2), derivative=False)
layer_3 = np.dot(layer_2, w_3)
return layer_0, layer_1, layer_2, layer_3

# Fill in backpropagation
def back_propagation(y, w_1, w_2, w_3, layer_0, layer_1, layer_2, layer_3):
    error = layer_3 - y

    # derivative of the error with respect to w_3
    layer_3_delta = np.dot(layer_2.T, error)

    # derivative of the error with respect to w_2
    d_layer_2 = np.dot(error, w_3.T) * sigmoid(layer_2, derivative=True)
    layer_2_delta = np.dot(layer_1.T, d_layer_2)

    # derivative of the error with respect to w_1
    d_layer_1 = np.dot(d_layer_2, w_2.T) * sigmoid(layer_1, derivative=True)
    layer_1_delta = np.dot(layer_0.T, d_layer_1)

    return layer_1_delta, layer_2_delta, layer_3_delta

# Cost function
def cost(X, y, w_1, w_2, w_3, lambda):
    N, d = X.shape
    a1,a2,a3,a4 = feed_forward_propagation(X,y,w_1,w_2,w_3,lambda)
    regularization = (lambda) * ((np.linalg.norm(w_1)** 2) +( np.linalg.norm(w_2)
→** 2) + (np.linalg.norm(w_3)** 2))
    return regularization + np.linalg.norm(a4[:,0] - y,2) ** 2 / N

# Define SGD
def SGD(X, y, w_1, w_2, w_3, lambda, learning_rate, batch_size):
    # Complete here:

    return w_1, w_2, w_3

# Define SVRG here:
def SVRG(X, y, w_1, w_2, w_3, lambda, learning_rate, T):
    # Complete here:

    return w_1, w_2, w_3

# Define GD here:
def GD(X, y, w_1,w_2,w_3, learning_rate, lambda, iterations):
    # Complete here:
    for i in range(iterations):
        # Forward pass

```

```

        layer_0, layer_1, layer_2, layer_3 = feed_forward_propagation(X, y, w_1,
↪w_2, w_3, lambda)

        # Backward pass
        d_w_1, d_w_2, d_w_3 = back_propagation(y, w_1, w_2, w_3, layer_0,
↪layer_1, layer_2, layer_3)

        # Regularization
        d_w_1 += lambda * w_1
        d_w_2 += lambda * w_2
        d_w_3 += lambda * w_3

        # Update weights
        w_1 -= learning_rate * d_w_1
        w_2 -= learning_rate * d_w_2
        w_3 -= learning_rate * d_w_3

    return w_1, w_2, w_3

# Define projected GD here:
def PGD(X, y, w_1,w_2,w_3, learning_rate, lambda, iterations, noise):
    # Complete here:

    return w_1, w_2, w_3

# Define BCD here:
def BCD(X, y, w_1,w_2,w_3, learning_rate, lambda, iterations):
    # Complete here:

    return w_1, w_2, w_3

```

```
[5]: y_train
```

```
[5]: array([ 1., -1., -1., ..., -1., -1.,  1.], dtype=float32)
```

```

[ ]: # Should be a hyperparameter that you tune, not an argument - Fill in the values
parser = argparse.ArgumentParser()
parser.add_argument('--lambda', type=float, default=0.01, dest='lambda')
parser.add_argument('--w_size', type=int, default=32, dest='w_size')
parser.add_argument('--lr', type=float, default=0.01)
parser.add_argument('--iterations', type=int, default=100)

#args = parser.parse_args()
args, unknown_args = parser.parse_known_args()

```

```

# Initialize weights
w_1 = initialize_w(X_train.shape[1], args.w_size)

w_2 = initialize_w(args.w_size, args.w_size)

w_3 = initialize_w(args.w_size, 1)

# Get iterations
iterations = args.iterations
# Define plotting variables
fig, ax = plt.subplots(2, 1, figsize=(16, 8))

# Define the optimizers for the loop
optimizers = [

    {# Fill in the hyperparameters
      "opt": GD(
        X_train, y_train, w_1, w_2, w_3, learning_rate=args.lr,
        lambda=args.lambda, iterations=iterations),
      "name": "GD",
      "inner": None
    },
]

```

```

[1]: # Should be a hyperparameter that you tune, not an argument - Fill in the values
parser = argparse.ArgumentParser()
parser.add_argument('--lambda', type=float, default=, dest='lambda')
parser.add_argument('--w_size', type=int, default=, dest='w_size')
parser.add_argument('--lr', type=float, default=)
parser.add_argument('--iterations', type=int, default=)

args = parser.parse_args()

# Initialize weights
w_1 = initialize_w(X_train.shape[1], args.w_size)

w_2 = initialize_w(args.w_size, args.w_size)

w_3 = initialize_w(args.w_size, 1)

# Get iterations
iterations = args.iterations
# Define plotting variables
fig, ax = plt.subplots(2, 1, figsize=(16, 8))

# Define the optimizers for the loop

```

```

optimizers = [
    {# Fill in the hyperparameters
      "opt": SGD(X_train, y_train, w_1, w_2, w_3, args.lambda, args.lr,
        ↪batch_size),
      "name": "SGD",
      "inner": # Fill in
    },
    {# Fill in the hyperparameters
      "opt": SVRG(X_train, y_train, w_1, w_2, w_3, args.lambda, args.lr),
      "name": "SVRG",
      "inner": # Fill in
    },
    {# Fill in the hyperparameters
      "opt": GD(
        X_train, y_train, w_1, w_2, w_3, learning_rate=args.lr,
        lambda=args.lambda, iterations=iterations),
      "name": "GD",
      "inner": # Fill in
    },
    {# Fill in the hyperparameters
      "opt": PGD(
        X_train, y_train, w_1, w_2, w_3, learning_rate=args.lr,
        lambda=args.lambda, iterations=iterations, noise=),
      "name": "PGD",
      "inner": # Fill in
    },
    {# Fill in the hyperparameters
      "opt": BCD(
        X_train, y_train, w_1, w_2, w_3, learning_rate=args.lr,
        lambda=args.lambda, iterations=iterations),
      "name": "BCD",
      "inner": # Fill in
    }
]

```

```

File "/var/folders/l9/nl6ry63s0m3_35qt7mrjv0fh0000gn/T/ipykernel_22324/
↪3547870537.py", line 3
    parser.add_argument('--lambda', type=float, default=, dest='lambda')
                  ^

```

SyntaxError: invalid syntax

```
[ ]: # Run the iterates over the algorithms above
```

```

for opt in optimizers:
    #

```

```
# Fill in
```

```
[ ]: # Plot results
ax[0].legend(loc="upper right")
ax[0].set_xlabel(r"Iteration", fontsize=16)
ax[0].set_ylabel("Loss", fontsize=16)
ax[0].set_title("CA3 - Training a deep neural network for the power consumption_
↳Dataset")
ax[0].set_ylim(ymin=0)

ax[1].legend(loc="upper right")
ax[1].set_xlabel(r"Time [s]", fontsize=16)
ax[1].set_ylabel("Loss", fontsize=16)
ax[1].set_ylim(ymin=0)

plt.savefig("power.png")
```